Efficient End-Host Architecture for High Performance Communication Using User-level Sandboxing

Xin Qi, Gabriel Parmer, Richard West, Jason Gloudon and Luis Hernandez

Computer Science Department
Boston University
Boston, MA 02215
{xqi, gabep1, richwest, jgloudon, lmh03}@cs.bu.edu

Abstract

Current low-level networking abstractions on modern operating systems are commonly implemented in the kernel to provide sufficient performance for general purpose applications. However, it is desirable for high performance applications to have more control over the networking subsystem to support optimizations for their specific needs. One approach is to allow networking services to be implemented at user-level. Unfortunately, this typically incurs costs due to scheduling overheads and unnecessary data copying via the kernel. In this paper, we describe a method to implement efficient application-specific network service extensions at user-level, that removes the cost of scheduling and provides protected access to lower-level system abstractions. We present a networking implementation that, with minor modifications to the Linux kernel, passes data between "sandboxed" extensions and the Ethernet device without copying or processing in the kernel. Using this mechanism, we put a customizable networking stack into a user-level sandbox and show how it can be used to efficiently process and forward data via proxies, or intermediate hosts, in the communication path of high performance data streams. Unlike other user-level networking implementations, our method makes no special hardware requirements to avoid unnecessary data copies. Results show that we achieve a substantial increase in throughput over comparable user-space methods using our networking stack implementation.

1. Introduction

Requirements for high performance networking extend far beyond the support provided by most modern operating systems. Specifically, abstractions such as BSD sockets and kernel-based network protocols are common to modern systems, but they are not tailored to the needs of high performance applications. These mechanisms are necessarily general so that they can provide fair, consistent, and simple abstractions of the base hardware to all processes. With this generality, fine grained control is sacrificed. For instance, there is little support for using efficient custom communication protocols in common systems. These generic mechanisms do not provide the power to efficiently utilize modern networking systems such as Gigabit Ethernet [21], ATM [11], and Myrinet [6].

To efficiently use many of the modern high throughput, low latency networking systems, specialized approaches must been taken that depart from the traditional operating system abstractions. Streamlined network processing stacks, zero-copy data movement, and asynchronous network processing are now seen as necessary for the highest degree of networking performance [24, 25]. A minimal, preferably application-specific networking path is important to reduce the latency involved in communication. For example, zero-copy eliminates any superfluous memory usage, while execution at interrupt time avoids unnecessary scheduling overheads.

Many systems have been devised that combine any number of these optimizations to provide a platform for high demand communication [9, 18, 24]. Most of these systems, however, require intrusive changes to their host kernel and specific functionality built into the hardware of the network interface to achieve enhanced performance. These are acceptable costs when the environment is controlled, as is the case in a research or government lab for scientific applications, but not feasible on the scale of the Internet where COTS systems must be considered.

An efficient mechanism is required with which an application can receive more control of the underlying hardware while still maintaining safety and isolation. Minimal impact on the code base of the kernel of the host system is certainly desirable. This mechanism can be provided by extensible systems.

Much work has been done in the area of extensible systems to provide specialized services to applications whose requirements go beyond the control and abstractions provided by the default interfaces of the host kernel. An extensibility mechanism strives to allow applications to execute code in a manner not normally provided, while maintaining safety. Some systems allow this application extension to be executed in the context of the kernel while still disallowing that code to modify the kernel or disrupt other processes [4]. Others implement minimal kernels which provide low-level interfaces and export much of the functionality and policy to the user level [10]. Linux provides the ability for trusted code to link directly into the address space of the kernel, without protection. Each of these mechanisms permit a greater degree of control of the hardware than would be otherwise available.

Using our extensibility mechanism called *user-level sandboxing* [27], we are able to provide access to lower-level abstractions such as interrupt-time execution within a bottom-half, and regulated control over packets transferred between memory and the network interface using DMA. Execution in the context of a bottom half enables the lowest possible latency for packet processing by avoiding scheduling overheads. Increased control of the network interface card (NIC) allows for zero-copy communication which reduces memory bus usage. Even with these privileged capabilities, the extension executes at user-level without access to the kernel address space. Further, because it executes in the application domain, the code may link with user-level libraries.

In summary the contributions of this paper center around support for efficient user-level implementations of network service extensions. The aim is to provide a means by which high performance distributed computing applications can customize network services for their specific needs. We leverage our ongoing work on *user-level sandboxing* to support the implementation of a network subsystem in user-space that avoids unnecessary intervention of the kernel. Moreover, our approach allows high performance applications to communicate with the network interface without: (1) the need to copy data via the kernel, and (2) scheduling overheads. This is achieved on commonly available hardware. We compare various implementations of a networking stack, traditionally implemented in the kernel, that forward data between end-hosts in a distributed system. This scenario would be applicable for efficient peer-to-peer routing of high bandwidth data streams, or in situations where a proxy server is handling remote procedure calls. Results show our system is flexible enough to allow applications to customize networking services for their specific needs, while providing efficient throughput comparable to kernel-level methods of forwarding network data.

The paper is organized in the following manner: Section 2 briefly describes the sandboxing mechanism required for our user-level networking services. Section 3 then discusses issues involved in the implementation of a networking stack in a user-level sandbox. This is followed by Section 4 that compares the performance of various networking implementations. Finally, related work is described in Section 5, followed by conclusions and future work in Section 6.

2. User-level Sandboxing

Our base architecture for extensibility is *user-level sandboxing*. For a full description of this mechanism, including a performance analysis, see our Sandboxing paper [27]. What follows is a an overview of all relevant information to our user-level networking scheme.

User-level sandboxing modifies the address space of all processes, or logical protection domains, to contain one or more shared pages of virtual addresses. The virtual address range shared by all processes provides a sandboxed memory region into which extensions may be mapped. Under normal operation, these shared pages will be accessible only by the kernel. However, when the kernel wishes to pass control to an extension, it changes the privilege level of the shared page (or pages) containing the extension code and data, so that it is executed with user-level capabilities. This prevents the extension code from violating the integrity of the kernel. The extension code itself can run in the context of *any* user-space process, even one that did not register the extension with the system, therefore eliminating scheduling overheads.

There is potential for corrupt or ill-written extension code to modify the memory area of a running process. To guard against this, we require extension code registered with the system to be written by a trusted programmer. By virtue of running at user-level, the kernel itself is always shielded from any extension software faults.

2.1. Hardware Support for Memory-Safe Extensions

Our approach assumes that hardware support is limited to page-based virtual memory (i.e., processors with an MMU). [1] This minimum hardware requirement is met by many processors made today. These relaxed requirements will allow wide deployment across a heterogeneous environment such as the Internet.

On many processors, switching between protection domains mapped to different pages of virtual (or linear) addresses requires switching page tables stored in main memory, and then reloading TLBs with the necessary address

[1] A series of caches, most notably one or more untagged translation look-aside buffers (TLBs) is desirable but not necessary.

translations. Such coarse-grained protection provided at the hardware-level is becoming more undesirable as the disparity between processor and memory speeds increases [22]. This is certainly the case for processors that are now clocking in the gigahertz range, while main memory is accessed in the 10^8Hz range. In practice, it is clearly desirable to keep address translations for separate protection domains in cache memory as often as possible. User-level sandboxing avoids the need for expensive page table switches and TLB reloads by virtue of the fact that the sandbox is common to all address spaces.

Traditional operating systems provide logical protection domains for processes mapped into separate address spaces, as shown in Figure 1(a). With user-level sandboxing (Figure 1(b)), each process address space is divided into two parts: a conventional process-private memory region and a shared, but normally inaccessible, virtual memory region. The shared region acts as a sandbox for mapped extensions. Kernel events, delivered by upcalls to sandbox code, are handled in the context of the current process, thereby eliminating scheduling costs.

2.2 Sandbox Regions

In our current implementation, the sandbox consists of two $4MB$ regions of virtual memory that are identically mapped in every address space to the same physical memory. For convenience, the two regions are assigned to adjacent (extended) page frames. That is, regions employ the page size extensions supported by the Pentium processor and each occupy one 4 megabyte page directory entry [2].

The Sandbox region is permanently assigned read-write permission at kernel-level, but by default is inaccessible at user-level. The region can be made accessible to user-level by toggling the user/supervisor flags of its page directory entry and invalidating the relevant TLB entries. This is only allowed when an upcall occurs from the trusted kernel.

2.3 Sandbox Threads

One design consideration is whether or not to allow threads to execute in the sandbox. If code in the sandbox is allowed to invoke system calls, it is possible for an extension registered by one process to block the progress of another process. For example, if process p_i registers an extension e_i that is invoked at the time process p_j is active, it may be possible for e_i to affect the progress of p_j by issuing 'slow' system calls. A currently available solution which allows extensions to use blocking system calls is to promote them to a thread. Scheduling this thread will have costs roughly comparable to the scheduling of a kernel thread [27].

[2]The 32-bit x86 processor uses a two-level paging scheme, comprising page directories and tables.

2.4 Pure Upcalls

Traditionally, signals and other such kernel event notification schemes [2, 16] have been used to invoke actions in user-level address spaces when there are specific kernel state changes. Unfortunately, there are costs associated with the traversal of the kernel-user boundary, process context-switching and scheduling. The aim is to implement an upcall mechanism with the speed of a software trap (i.e., the mirror image of a system call), to efficiently vector events to user-level where they are handled by service extensions, in an environment that is isolated from the core kernel. It should be noted that sandbox extensions cannot be invoked other than via the trusted kernel. We call this type of an invocation where the extension code is run at the user-level a *pure upcall*.

Of primary benefit to any high performance network communication framework is the ability to entirely avoid any scheduling thus processing packets immediately when they arrive. Therefore, any extension written for this purpose can be executed independent of any scheduling as a pure upcall.

3 User-level Networking

The motivation for this work is to support high performance distributed applications that require system services configured for their specific needs. In meeting this goal, we have implemented an entire networking subsystem in a user-level sandbox that can be customized to support application-specific protocols and services. This section describes the issues involved in its construction.

- Memory management: Memory resources in the sandbox area must be managed efficiently to oversee packet placement and to ensure all allocations occur within the sandbox. `malloc`, the memory allocation tool provided with glibc, is not satisfactory for this high performance role. It is beneficial to have a slab allocator [7] that has apriori knowledge of objects such as packet descriptors (`sk_buff_heads`).
- Kernel Bypass: An abstraction for passing control to the extension during a bottom half asynchronous execution unit must exist. This abstraction must include hooks which are executed during packet reception in the kernel to trigger execution in the sandbox. This allows the network execution path in the kernel to be removed in favor of the more specialized extension's code. The abstraction allows insertion of customized network code into the critical networking path. This is

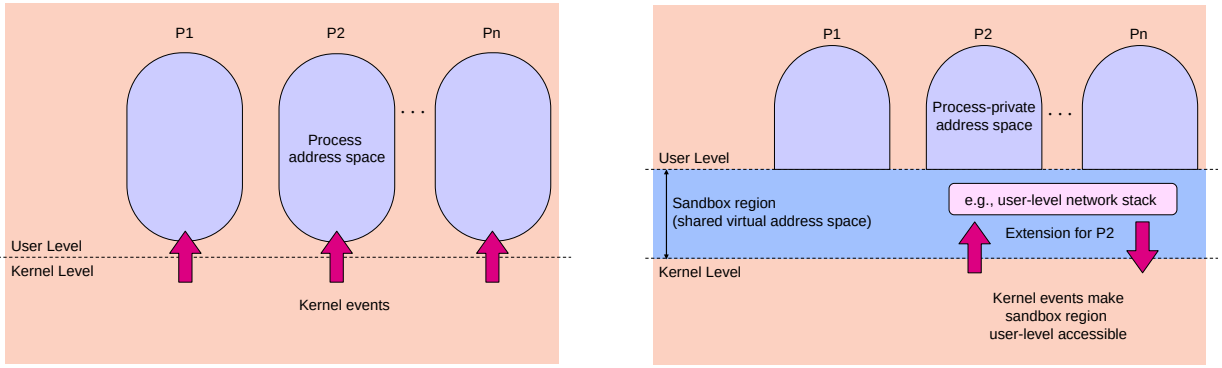

Figure 1. (a) Traditional view of logical protection domains, each mapped to a separate address space. (b) With user-level sandboxing each process address space has a shared virtual memory region into which extensions are mapped. For example, an extension might support a network stack that is activated by kernel events when packets arrive on a network interface.

seen as primary to obtaining high networking performance [24, 9].

- NIC interaction: An interface between the NIC and the sandbox allowing packets to be received into and sent from the sandbox directly using DMA. This is essential in data stream processing because the packets will be very large, and we will benefit from the zero-copy afforded to us by this interface.

In consideration of all of these criteria, User-mode Linux (UML) [23] was chosen as the platform to insert into the sandbox. UML is, in essence, the Linux kernel ported to user-space. It is a type of virtual machine that executes as an application on top of a host Linux kernel. All of the hardware of a real machine is emulated using the system-call, signaling, and ptrace interfaces of the host kernel. It provides all of the services that Linux itself does including, most importantly, memory allocation, a modular device interface, and a fully functional, well tested, efficient and modular networking stack. Extensions to UML's networking code can be written, and loaded into the sandbox with it. Furthermore, because UML is a user-space application, it is much easier to port to the sandbox than a normal kernel. UML satisfies the criteria of memory management with its internal `kmalloc` interfaces. Using UML allows us to manage the memory resources in the sandbox with the same efficiency and granularity as in the host kernel. A benefit of this decision is that memory is strictly controlled within the $8M$ boundary.

The design choice to use UML makes the conditions for kernel bypassing easily satisfiable. Driver abstractions exist within the Linux kernel, and therefore in UML, to receive and transmit packets. These abstractions make it simple to write a driver in UML that functions as an intermediary between the host kernel bottom half and the extension networking stack. It is only because the sandbox is mapped across all virtual address spaces that it is possible to support efficient bottom half execution. Typically, it would not be efficient or desirable to execute a user-level handler in the context of a bottom half because it is impossible to guarantee that a process's virtual address space will be currently loaded. Executing the process's code could require a context switch. This is a costly operation to do for every interrupt caused by network hardware.

UML, however, does provide abstractions and mechanisms that are unnecessary in a sandbox environment. Namely, the UML analogy of user-level processes are superfluous to our purposes of providing a customizable networking stack in the sandbox. After implementing all changes required to make UML run in the sandbox without these virtual user-level processes and with a specific driver to interface with the host kernel, no more than a few hundred lines of code were altered.

Because extensions running in the sandbox cannot issue privileged instructions, they cannot directly modify or influence the networking hardware to copy packets directly to and from the sandbox using DMA. Instead, the host kernel and sandbox interface through a well defined set of communication channels to pass the desired memory location for packet arrival or transmission. In this way, all protected instructions and all kernel memory remains protected from sandbox extensions. Independent of which process is running at any given moment, the networking card can DMA directly to or from the sandbox region. This type of communication is only possible because the sandbox exists in every process virtual address space. Consequently, no special hardware requirements are placed on the networking hardware. DMA is used in all our examples because its deployment on modern networking cards is so ubiquitous, but there is no innate limitation which restricts us to DMA. This is an important point because the sandboxing user-level net-

working system is intended to be deployed Internet-wide and must be as hardware heterogeneous as possible.

In this manner the NIC interaction criteria for providing a high performance, user-level, networking interface is fulfilled. This interface is leveraged to provide a minimal copy capability where packets coming in from the network can be copied using DMA directly into the sandbox and, after processing by the customizable stack, can be copied directly back onto the network. This ability has been used to provide a direct proxying service within the sandbox. Packets of a data stream can be routed without copying (from the CPU's view) at the transport level of the customized networking stack along a backbone network.

Demultiplexing: The demultiplexing of packets is one of the challenges when developing a user-level networking stack. Other technologies rely on programmable NICs which have apriori knowledge of the destination of incoming packets so they can transfer them directly to the correct destination. We do not take this approach because we wish to target non-specialized NICs. Instead, a light weight classifier can be written either in the lowest levels of the host kernel or in the sandbox, which then dispatches packets appropriately. In either case, all incoming packets must still be allocated and transfered (perhaps via DMA) to the sandbox address range. The sandbox networking scheme is not intended to be an architecture for the efficient processing of *all* packets, only those which extensible code was written to handle. This is not a serious limitation because it is unlikely that *every* networking stream will require high-performance communication; only a subset will need such efficiency.

3.1. Control Flow With Sandboxed Networking

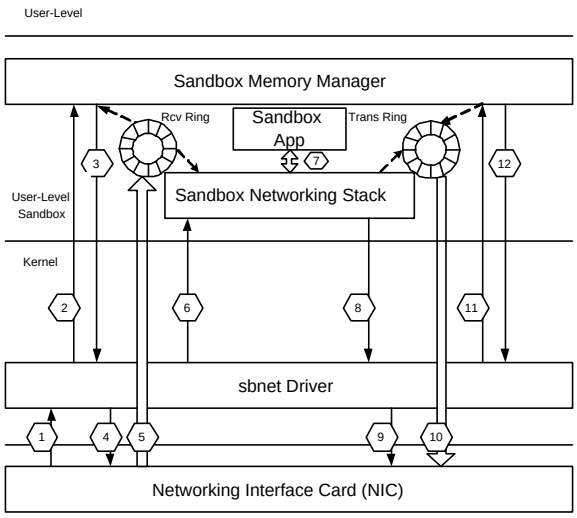

Figure 2. User-level networking in a sandbox.

The execution path of sandboxed networking is demonstrated by Figure 2. To support specialized interfaces with the sandbox extension, we modified the host's networking driver (referred to as the sbnet driver.)

1. When a packet is received on the NIC, an interrupt (top half) is invoked in the sbnet driver. This is a basic notification that a packet is ready and a minimal amount of processing is undertaken. No modifications to the top half of the default driver are made so that it remains as efficient as possible.
2. Interrupts are re-enabled and bottom half execution proceeds in the sbnet driver. We then allocate space in the sandbox by performing an upcall that directly invokes the sandbox memory manager(`kmalloc`).
3. The return value of this allocation is passed to the sbnet driver. To ensure that the extension in the sandbox is not being malicious, a check is performed to verify that the memory location is within the sandbox virtual address space.
4. The sbnet driver informs the NIC of the location into which it can DMA the packet. Because this sbnet driver is executed in the kernel domain, it has full I/O permissions and can communicate safely with the NIC.
5. The NIC copies the received packets into the allocated sandbox memory using DMA.
6. After the new packet is resident in the sandbox, a pure upcall is made and the networking stack in the extension is invoked. Note that this is still in the context of a bottom half, so execution is unaffected by host scheduling. Recall that when a pure upcall is triggered, the handler runs with user permissions.
7. At points in the configurable networking stack, application specific handlers can execute. We provide an application that performs transport level routing to directly forward packets to another end-host system.
8. After the network stack's processing is complete, it may wish to transmit a packet and returns its address. Upon return from the pure upcall initiated in step 6, we resume execution in the kernel.
9. Because full I/O permissions are restored, the NIC is notified. of the packet it should transmit.
10. The NIC uses DMA to retrieve and send the packet onto the network.
11. After the DMA is complete, the sbnet driver notifies the sandbox extension that it can free the memory formerly taken up by the packet. The memory manager is invoked to free it.
12. Upon return from this pure upcall, we have completed the bottom half and can return to the previous execution thread. Though this path seems complex, it is highly optimized and gets excellent performance as is shown in the results.

4. Experimental Evaluation

Our experiments are executed on a cluster of 8 IBM x series 305 e-servers, each with a tigon3 gigabit Ethernet card, interconnected by a gigabit Ethernet switch. Each machine has 2.4G Pentium IV CPUs and 1024M RAM. Our code is based on the Linux 2.4.20 kernel.

4.1. UDP Forwarding

We use a simple *UDP forwarding agent* to test the performance characteristics of our implementation. This application forwards UDP packets for specific streams of data from one host, which we will call *A*, through the forwarding host, *B*, to the receiving host, *C*. Such a generic application could be used to route data along a P2P overlay, or over a Grid. The forwarding would typically be directed and arbitrated by routing protocols, perhaps defined by a hashing algorithm for a P2P network [3], or by a library such as the Globus toolkit [12] for a grid. This application level routing protocol could be built into the sandbox, but, to best demonstrate and stress the high performance capabilities our method enables, we conducted our experiments using only the *UDP forwarding agent*. An application analogy of this service is *squid* [20].

To generate traffic and measure throughput and jitter, we used the *Iperf* [15]. Iperf generates packets at the source, *A*, and measures the perceived throughput at the destination, *C*. The perceived throughput can be less than the amount sent because packets might be lost in transit, at *B* for instance. All bandwidth figures shown in this section are of the perceived throughput. Iperf also measures the jitter of the arrival time of the packets. That is, it measures the deviation from the average transfer time of each of the packets and it maintains a running total of this average jitter.

4.2. Comparison of Networking Implementations

The experiments are broken into groups, each demonstrating a specific aspect of the benefit of the work. The first experiment focuses on the ease of porting an application to the sandbox while obtaining benefits in the form of performance.

User-Level Networking: Figure 3 shows a comparison between the capability of an UDP forwarding agent running on host *B* in User Mode Linux as an application in user-space and UML as an extension in the sandbox. MTU sized packets were routed from *A* to *C* and the throughput was noted. The UML forwarding agent operates in the same way as our sandboxed network scheme: All processing and routing is done in the UML equivalent of a bottom half. The two main differences between the test cases are that UML must be scheduled, and that the sandbox code can make use of DMA thus reaping zero-copy benefits. The sandbox extension code is essentially unchanged besides the fact that we disable UML's user-level processes in the sandbox, as mentioned before.

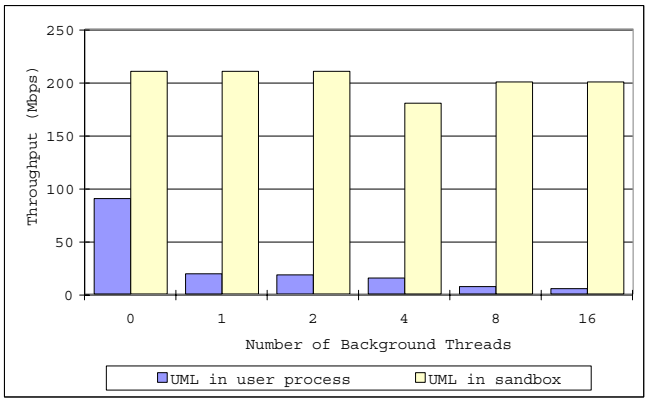

Figure 3. Throughput comparison of UML in a sandbox versus user-process.

Figure 3 shows the throughput corresponding to both of these cases. Background processes are run to measure the effect of system load on performance. These background processes are simple while loops with minimal working-set sizes. It is important to measure the throughput of our methods with background processes because we target COTS systems on the Internet that could be performing other tasks concurrently.

The results show that with no background processes, the sandbox forwarding agent demonstrates an improvement of 130% over the user-level UML. Further, as the number of processes increases, application-level UML does consistently worse as scheduling impedes its progress. The extension does not suffer from this degraded progress because it is executing as an asynchronous unit, independent of any scheduling. It is evident from the results that installing user-mode code into the sandbox with minimal porting effort can improve the throughput significantly.

The execution semantics of the sandbox allow for extension code to execute that is not controlled by mechanisms that exist within the kernel to provide fairness such as CPU scheduling. However, the sandbox is an architecture designed for *trusted* code which is to be inserted by a trusted user. Furthermore, this asynchronous execution environment together with efficient interaction with the networking card, make a good environment for high performance computing where high throughput and low latency are desirable.

User- versus Kernel-Level Networking: The former experiment demonstrated that with minimal porting effort, a user-level code base can be made to execute in the sandbox,

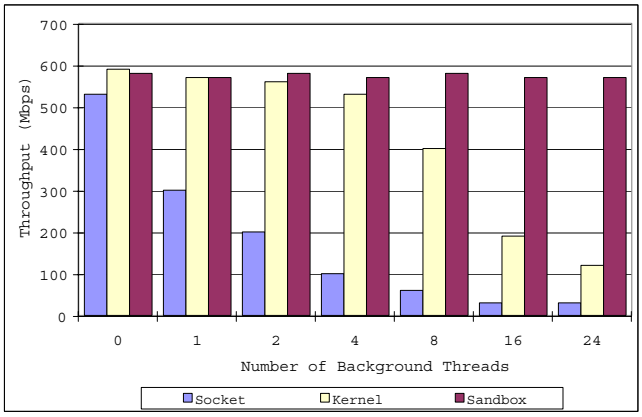

Figure 4. Throughput comparison of an optimized sandbox stack versus both user-level sockets and kernel implementations.

thus reaping performance benefits. However, it did not provide a good comparison with which to demonstrate the raw power of the sandbox services. UML is afflicted by some degree of virtualization costs making it a weak candidate for high-performance communication.

The next test, therefore, focuses on the performance aspects of our user-level networking scheme. We further optimize the extension to fully take advantage of the sandboxed environment. It is acceptable, using pure upcalls and the sandbox, to make non-blocking system calls, but it can affect performance. UML normally disables interrupts to ensure synchronization. To do this it makes system calls which disable signals that are used to virtualize the asynchronous events of a host kernel. Only one bottom half (softirq in Linux) can execute per processor at any one time. By utilizing this synchronization already provided by the kernel, the synchronization related system calls within UML are unnecessary.

After removing synchronization system calls, performance increases by nearly a factor of three as can be seen in Figure 4. We compare the throughput of the sandbox networking stack with two other implementations: that of a simple forwarding agent within the kernel that uses a kernel thread to send from one socket to another without copying the data, and with that of a user-level application that simply reads the data off of a socket and writes it to the next. The latter application should be viewed as the most efficient possible middle-ware solution.

The same testing environment as before is maintained and maximum throughput from *A* to *C* through *B* is measured with a certain number of background processes. One can see that the sandboxed networking stack's bandwidth remains nearly constant as the number of processes increases while the other approaches do not. Remember that the kernel routing uses a kernel thread, which is a schedulable entity so it has to compete for CPU time. Even with no background processes, our user-level networking does better than the minimal user-level application and only slightly worse than the kernel itself.

4.3. Transfer Time Jitter

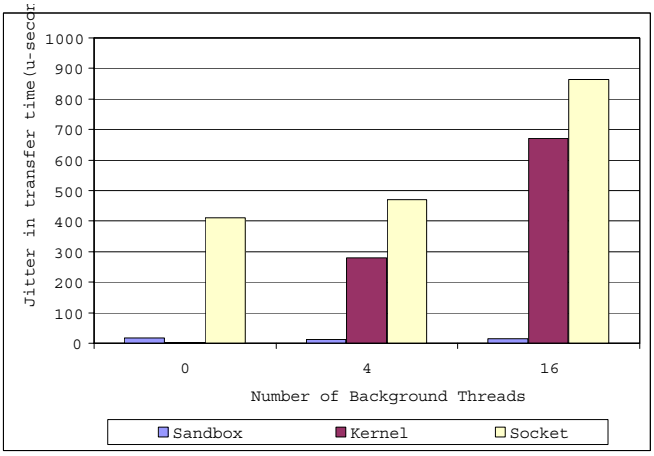

Figure 5. Maximum jitter in transfer time.

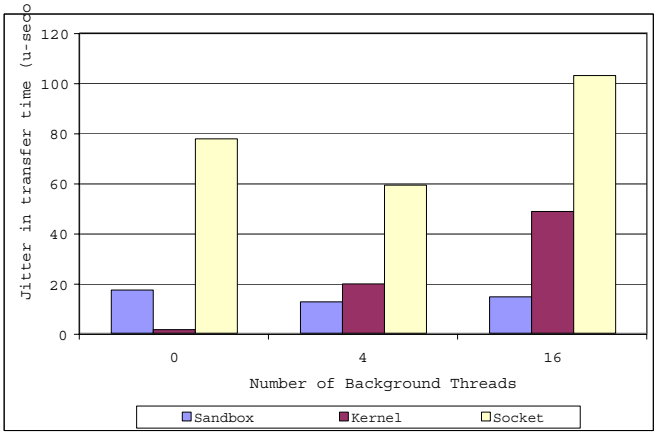

Figure 6. Average jitter in transfer time.

Average jitter measures the variation of the transit time of the packets over a period of time. Maximum jitter, often more important, measures the maximum of the deviation from the average transfer time. The reduction or elimination of jitter is especially important in environments which require QoS constraints to be met. If network performance is unpredictable (i.e., high jitter is present) then guaranteeing QoS constraints becomes increasingly difficult if not impossible.

Running in the context of bottom halves gives the sandbox pure upcall code the ability to immediately process each incoming packet, which results in a very small amount of variation in the transfer time of those packets. In contrast, the kernel and process-based forwarding agents must suffer from scheduling delays. The amount of deviation from the average transfer time is a function of the size of the run queue. Figure 5 and Figure 6 show that a nearly consistent amount of jitter is demonstrated by the Sandboxed networking scheme while the other two show degradation as the amount of background processes increases.

4.4. Microbenchmarks

Operation	Cost in CPU cycles
Null Pure Upcall	1370
Sandbox Packet Processing Time	6360
Kernel Packet Processing Time	4800

Table 1. Sandbox Overheads.

Table 1 shows microbenchmarks measured by using the real-time clock. The round-trip time for a pure upcall invoking a function that simply returns to the kernel is 1370 clock cycles. Due to various optimizations that avoid identifying the process responsible for registering an upcall function, this value improves upon the costs of upcalls in our earlier work [27]. In our original upcall implementation, it was necessary to identify the process that registered an upcall so that its credentials (including file descriptors) could be made available for use by an upcall handler. This is no longer necessary for our current work.

The second value in Table 1 measures the time it takes to process a packet using our user-level networking code. A measurement is taken upon reception of a packet and again when we transmit that packet. This overhead compares favorably to the cost of executing a network bottom half handler in the kernel. Hence, the overheads of using our sandboxing scheme do not impose excessive costs on the implementation of network services at user-level.

5. Related Work

Many researchers have explored various methods for high performance communication. For example, Active Messages [25] can be used to implement an efficient RPC [5] mechanism by running as handlers in the context of the currently active protection domain. This avoids scheduling and context switching overheads to implement network services, as does our approach using user-level sandboxing.

In effect, our work is similar to that of U-Net [24], Ethernet Message Passing (EMP) [18], and the Virtual Interface Architecture (VIA) [9], that all provide abstractions for user-level network implementation. In these alternative approaches to ours, the network interface card (NIC) is virtualized and multiplexed across applications. Additionally, if hardware permits, zero-copy capabilities are available. EMP requires programmable NIC interfaces to offload message processing to hardware and provide zero-copy capability. While U-Net and VIA are also able to advantage of advanced hardware, they can only run on non-programmable NIC cards at the cost of efficiency. In all cases, they find that application-specific network implementations offer substantial performance improvements over similar implementations that run on generic OS abstractions. The fine grained control of the network allows applications to optimize resource usage. Our work doesn't provide such a demultiplexing abstraction. Instead, it allows user-level extensions to run efficiently enough to be invoked as handlers for networking events.

There have been a number of related research efforts that focus on OS structure and extensibility, safety, and service invocation. Extensible operating systems research [19] aims at providing applications with greater control over the management of their resources. SPIN [4], for example, is an operating system that supports extensions written in the type safe Module-3 programming language. By using type safety and interface contracts to provide protection, extensions can be injected into the operating system and run at kernel level. In addition to being able to extend kernel functionality, like memory management and scheduling, they show their approach provides improved network latency and lower CPU utilization over user-level implementations of protocol forwarders and video servers. Our work attempts to bridge the performance gap between user and kernel-level network implementations evident in the SPIN experiments.

Transaction-based approach to system extensibility is employed by the VINO [17] operating system. VINO supports system extensions known as grafts that are written in C++. Since C++ is not type-safe and memory protection is an issue, grafts are run in the context of transactions, so that the system can be returned to a consistent state if a graft is aborted. We are currently working to provide CPU constrained execution for extensions running in the Sandbox, using techniques found in the VINO and SafeX [26] work. Such a method would allow us to avoid the situation where a extension executing as a bottom half uses more than a constrained amount of resources.

In contrast to safe kernel extensions, micro-kernels such as Mach [1] and Exokernel [10] offer a few basic abstractions, while moving the implementation of more complex services and policies into application-level components. By

separating kernel and user-level services, micro-kernels introduce significant amounts of interprocess communication overhead. This has caused micro-kernels to fall out of favor despite substantial reductions [14] in communication costs. Regardless of the overhead, a network implementation running on Exokernel achieved an average a factor of 2-5 improvement in throughput over conventional implementations [13]. Our approach differs in that it is aimed at extending existing systems, by slight modifications to the kernel, as opposed to employing an entirely new (albeit small) trusted kernel.

Finally, observe that our work is not to be confused with user-level resource-constrained sandboxing [8], by Chang, Itzkovitz and Karamcheti. Their work focuses on the use of sandboxes to enforce quantitative restrictions on resource usage. They propose a method for instrumenting an application, to intercept requests for resources such as CPU cycles, memory and bandwidth. As a result, they are able to control the allocation of such resources in a predictable manner. The emphasis of our work is to develop an efficient execution environment at user-level for kernel extensions and system services, regardless of which address space is active at the time extension code is invoked.

6. Conclusions and Future Work

We have devised a high performance networking scheme that, utilizing the sandbox extensibility service, can provide levels of throughput higher than those of middleware while coming close to the kernel's native throughput while still utilizing a fully-featured networking stack. The user-level sandboxing scheme allows networking extension code to safely and efficiently access and influence lower-level abstractions such as bottom halves and the network hardware. It is through these exposed abstractions that such high performance is available. The sandboxing mechanism itself allows for customizable network processing units or stacks to execute with user-level permissions. These extensions, therefore, cannot compromise the integrity of the kernel's address space and are well encapsulated. Moreover, this efficiency and safety are provided without cumbersome hardware requirements. The only requisite is that of a paging memory management unit which is ubiquitous in modern computers. It follows that wide deployment across a heterogeneous environment such as the Internet is possible and desirable.

Many aspects of this work can be expanded. We are exposing the ability to deploy efficient network services to a trusted user, for the benefit of implementing functionality specific to high performance applications. Future work involves the implementation of sandboxed services that have isolated and controlled resource usage. For example, an end-system service extension used for the purposes of processing and forwarding high bandwidth data should not prevent a user of the system from performing other tasks by consuming all available resources. Other future work involves exposing hardware devices to extensions executing in a user-level sandbox, thereby supporting user-configurable device drivers. Finally, to make our sandboxing system truly portable, we intend to eliminate the need to make any modifications to the core kernel. In fact, we are currently investigating a number of binary-rewriting techniques to achieve this objective. This will enable our sandboxing system to be easily deployed on multiple hosts in a distributed environment.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, pages 93–112, Atlanta, GA, USA, July 1986.

[2] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[3] Beep: The application protocol framework: http://www.beepcore.org.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[6] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigbit-per-second local area network. In *IEEE Micro*, pages pages 29–36, 1995.

[7] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[8] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium, 2000. Seattle, WA*, Seattle, WA, 2000.

[9] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. In *IEEE Micro*, 1998.

[10] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

[11] A. Forum. *ATM User-Network Interface Specification Version 3.0*. Prentice Hall, Englewood Cliffs New Jersey, 1993.

[12] I. Foster and C. Kesselman. Globus: A toolkit-based architecture. *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278, 1999.

[13] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, 2002.

[14] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. Ther performance of μ-kernel-based systems. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*. ACM, October 1997.

[15] Iperf version 1.7.0: http://dast.nlanr.net/projects/iperf/.

[16] J. Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2001.

[17] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[18] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *High Performance Networking and Computing, SC2001*, 2001.

[19] C. Small and M. I. Seltzer. A comparison of os extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[20] Squid web proxy cache: http://www.squid-cache.org/.

[21] G. e. alliance. ieee 802.3z. the emerging gigabit ethernet standard., 1997.

[22] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

[23] User-mode linux kernel: http://user-mode-linux.sourceforge.net/.

[24] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53. ACM, December 1995.

[25] T. von Eicken, D. E. Culler, S. C. Golsdtein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. Report UCB/CSD 92/675, University of California, Berkeley, March 1992.

[26] R. West and J. Gloudon. 'QoS safe' kernel extensions for real-time resource management. In *the 14th EuroMicro International Conference on Real-Time Systems*, June 2002.

[27] R. West and J. Gloudon. User-level sandboxing: a safe and efficient mechanism for extensibility. Technical Report 2003-014, Boston University, June 2003.